Task 6.1 TRUE. Kruskal's algorithm works with negative edge weights, because it only relies on an ordering principle of edge weights, not their values. This is also why adding a constant to each edge weight doesn't change the output of Kruskal's algorithm. Hence, another way of thinking about this would be that Kruskal's algorithm is known to be correct with positive edge weights, and when we add a large enough positive constant to make all edges positive, the output is the same as for the graph with negative edge weights. Therefore, Kruskal's algorithm must produce the correct output for a graph with negative edge weights without any modifications.

Task 6.2 TRUE. If we negate all of the edges, the ordering of the edges is flipped, so the maximum edges will be selected first. Since an MST must be a tree, we cannot pick any extra edges, because we would create cycles. Another way of thinking about this is: if we minimize the sum of $|V| - 1$ negative edge weights, we are effectively maximizing the sum of $|V| - 1$ positive edge weights.

Task 6.3 FALSE. Consider a graph of two nodes connected by an edge of weight 3. The MST contains the graph's only edge, which is trivially its heaviest edge. A couple other counterexamples would be any graph with $|V| - 1$ edges, where the MST must contain all edges, as well as any graph where there is a node connected to only one other node by an edge with the heaviest weight, where the MST must contain this edge to span the graph.

Task 7.1 Given the source vertex A, these are possible DFS orderings:
{A, B, C, X, H, M}
{A, B, M, X, H, C}
{A, B, M, H, X, C}
{A, M, X, H, B, C}
{A, M, H, X, B, C}

Task 8.1 The asymptotic complexity of this version of heapsort is $O(n \log n)$. There are two loops which iterate through $O(n)$ nodes:

The first loop calls percolateUp each iteration, which has $O(\log n)$ runtime since the depth of the heap is n, so it has to perform $O(\log n)$ constant time swaps per iteration. While it is true that the nodes near the top are percolating up only a few levels, the $O(n)$ leaf nodes potentially have to percolate up all the way to the top, so the overall time complexity of the first loop is $O(n \log n)$. This could technically be improved by Floyd's algorithm, which percolates down instead of up, meaning the leaves have to do the least work, leading to a sum that is $O(n)$, but this fact is irrelevant because the runtime is constrained by that of the second loop.

The second loop calls percolateDown each iteration, which has $O(\log n)$ runtime since the depth of the heap is n, so it has to perform $O(\log n)$ constant time swaps per iteration. Though the size of the heap decreases as we place removed elements at the end, the average size is $\frac{n}{2}$, which is still $O(n)$. The additional swap also runs in constant time. Hence, we have $O(\log n)$ work for each of n iterations, leading to $O(n \log n)$ runtime for this loop.

Therefore, our overall runtime is $O(n \log n)$.

Task 9.1

Time Step	Visited	State of Queue
0	-	(0,s)
1	(0,s)	(1,a), (2,b), (3,c)
2	(0,s), (1,a)	(2,b), (3,c), (4,f)
3	(0,s), (1,a), (2,b)	(3,c), (4,f), (3,e)
4	(0,s), (1,a), (2,b), (3,c)	(4,f), (3,e)
5	(0,s), (1,a), (2,b), (3,c), (3,e)	(4,f)
6	(0,s), (1,a), (2,b), (3,c), (3,e), (4,f)	(10,g), (6,h)
7	(0,s), (1,a), (2,b), (3,c), (3,e), (4,f), (6, h)	(9,g)
8	(0,s), (1,a), (2,b), (3,c), (3,e), (4,f), (6, h), (9, g)	